# Interface Stability

Mark Kampe

## Interface Specifications

American History books like to credit Eli Whitney with the invention of manufacturing with interchangeable parts. Actually,

- Whitney's parts weren't all that interchangeable,
- he got the idea from a French gunsmith (Honore Blanc),
- the Swedish clock maker Christopher Polhem did a much better job with clock gears 50 years earlier.

But, be that as it may, interchangeable parts revolutionized, and in fact enabled modern manufacturing. The underlying concept is that there be specifications for every part. If each part is manufactured and measured to be within its specifications, then **any** collection of these parts can be assembled into a working whole. This greatly reduces the cost and difficulty of building a working system out of component parts.

The same principle applies to software. If you want to open a file on a Posix system, you use the **open** system call. There may be a hundred different implementations of open but this doesn't matter because they all conform to the same interface specification. The rewards for this standardization are:

a. Your work as a programmer is simplified, because you don't have to write your own file I/O routines.
b. Your program is likely to be easily portable to any Posix compliant system, because they all provide the same file I/O services.
c. Training time for new programmers is reduced, because everybody already knows how to use the Posix file I/O functions.

## The Criticality of Interfaces in Architecture

A system architecture defines the major components of the system, the functionality of each, and the interfaces between them. Clearly the component interface specifications are a critical element of any architecture. To the extent that these interfaces have been well considered and well defined, it should be possible to (at least semi) independently design and implement those components. In principle, any implementations that satisfy those functional and interface specifications should combine to yield a working system.

The converse is also true. To the extent that interfaces between components were poorly considered or specified, it becomes unlikely that independently developed components will work when combined together, and more likely that subsequent changes to one component will break others.

## The Importance of Interface Stability

We write contracts is so that everybody knows what will be expected of them, and what they can expect from the other parties to the agreement. Everybody will depend on you to uphold your obligations under the contract. A software interface specification is a form of contract:

- the contract describes an interface and the associated functionality.
- implementation providers agree that their systems will conform to the specification.
- developers agree to limit their use of the functionality to what is described in the interface specification.

And in return for this they get all the benefits described above.

Suppose that some architectural genius realized that Posix file semantics are too weak to describe the behavior of files in a distributed system, and that this could be solved by adding a new (required) parameter (for a call-back routine) to the Posix open call. What would happen?

- software written to the new semantics would have access to richer behavior, and would function better in cases of node and network failures.
- software written to the new semantics would not work on other Posix compliant systems.
- software written to the old semantics would not work on the new systems.
- customers (knowing nothing about this techno-feud) would be faced with inexplicable failures, and would start complaining to their software and system providers (none of whom were responsible for the change).
- programmers would be confused about which version of open they were using, and would probably get around the problem by writing their own file I/O packages ... a bunch of extra work, but at least it would protect them from future such acts of mischief.

And as a result of this, we would lose all of the benefits described above. When you promise somebody that you will do something (e.g. conform to a specified interface), and they depend on you (e.g. by writing code to that interface), and you don't follow through (e.g. make an incompatible change) ... problems are likely to ensue (e.g. failures, bug reports, product gets a bad reputation, going out of business, etc.).

# Interfaces vs. Implementations

Suppose that someone writes a handy library to efficiently provide reliable communication over an unreliable network, and I decide I want to use it. I download their development kit and start using it, and find it to be great. A few weeks into the project I realize that I need a feature that is not included in their documentation, but after reading their code I discover that the needed feature is actually available. I use it and my product is a great success.

Six months later, they release a new version of their reliable communications library, and my product immediately breaks. After a little debugging I discover that they have changed the undocumented code that I was depending on. It turns out that I had not designed my product to work with their *interfaces*. My product only worked with a particular *implementation* ... and implementations change.

People often put much more work into designing and maintaining their code than to clarifying and maintaining their documentation. This makes it tempting to reverse engineer the interface specifications from a provided implementation. But an interface should exist (and be defined) independently from any particular implementation. Confusing an implementation with an interface usually ends badly!

# Program vs. User Interfaces

Human beings are amazingly robust. We could change the text in dialog boxes, rearrange input forms, add new required input items, and rework all of the menus in a program, and many users would figure out the changes in a few seconds. This inclines many developers to be cavalier in making changes to user interfaces.

Code is nowhere nearly so robust. If I expect my second parameter to be a file name, and you pass me the address of a call-back routine instead, the best we can hope for is a good error message and a quick coredump with no data corruption.

If all users were developers, and only ran their own code, this might be just an irritation. We would get the core dump, track it down, figure out that some idiot had made this change, recode accordingly, and an hour later we'd be good as new. Unfortunately most people run code that they do not understand, and have no way to debug or fix. If a program breaks, all I can do is call support and complain. I am helpless. Users don't care which programmer goofed up. "I bought your product. It doesn't work. I'm taking my business elsewhere!".

If you are going to be delivering binary software to non-developers (that describes 99.999% of all software) you have to trust that what ever platform they run it on will correctly implement all of the interfaces on which your program depends.

The same argument applies, though not quite as strongly, to independent components in a single system. If components exchange services, and I make an incompatible change to the interfaces of one component, this has the potential to break other components in the same system. I can fix the other components in our system to work with the new changes but:

- this complicates the making of the change.
- we have to ensure that mismatched component sets are impossible.

# Is every interface carved in stone?

Absolutely not.

You are free to add features that do not change the interface specification (a faster or more robust implementation). In many cases you can add upwards-compatible extensions (all old programs will still work the same way, but new interfaces enable new software to access new functionality). There are a few ways to add incompatible changes to old interfaces without breaking *backwards compatibility*:

**Interface Polymorphism**

In old-school languages (like FORTRAN and C) a routine had a single interface specification. But more contemporary programming languages support polymorphism (different method signatures with different parameter and return types). If different versions of a method are readily distinguishable (e.g. by the number and types of their parameters), it may be possible to provide new interfaces to meet new requirements, while continuing to support the older interfaces for backwards compatibility with older applications.

**Versioned Interfaces**

Backwards comptibility requirements do not prevent us from changing interfaces; they merely require us to continue providing old interfaces to old clients. In many cases, it may be possible for an application to call out which version of an interface it requires:

- Java programs are labeled with the run-time-environment versions they require.
- Network protocol (e.g. RPC) sessions often begin with a version negotiation, where each side states the interface versions that it can support, so that a mutually acceptable version can be chosen.

This approach does permit incompatible interface changes, but still requires servers to support old as well as new interface versions.

But even if we have no ability to deliver multiple interface versions, this does not necessarily mean I can never make incompatible changes. It is not the case that all interfaces must be supported for all of time. It is merely necessary that the interface stability be understood, and adequate to achieve the product (or project) goals:

- It is common to provide alpha/evaluation access to proposed new services/interfaces. This is done with the understanding that the final interfaces are likely to be different. Such releases are provided purely for evaluation purposes. Understanding the likelihood of change, developers should only use them for prototyping, and should not base long-lived products on them.
- Many suppliers have interface stability guidelines for their releases. A typical example might be:
  - a micro-release (e.g. 3.4.1) includes only bug fixes.

    - a minor-release (e.g. 3.5) includes new functions, but us upwards compatible with the underlying major release version.
    - a major release (e.g. 4.0) is allowed to make incompatible changes to previously committed interfaces.

  Applications software develelopers can use this to determine which versions their software will run on, and customers can use this to decide whether or not they can safely upgrade to a new release.
- Many suppliers have make specific support commitments (e.g. we will support release 3.5 for five years from its initial availability date). This gives software developers and customers confidence that they will have a stable software platform for at least that long. At the end of that period, customers may have to choose between continuing to run on a no-longer-supported platform or moving to new versions of their mission-critical applications.

How stable an interface needs to be depends on how you intend to use it. What is important is that:

1. we be honest about our intentions and set realistic expectations.
2. we have a plan for how we will manage change.

# Interface Stability and Design

So, if we want to expose an interface to unbundled software with high quality requirements, we are not allowed to change it in non-upwards-compatible ways. That sounds like a bother, but if that is the rules, so be it. What has this got to do with architecture and design?

When we design a system, and the interfaces between the independent components, we need to consider all of the different types of change that are likely to happen over the life of this system. When we specify our external component interfaces we need to have high confidence that will be able to accomodate all of the envisioned evolution while preserving those interfaces.

- We might rearrange the distribution of functionality between components to create a simpler (and hence more preservable) interface between them.
- We might design features that we may never implement, just to make sure that the specified interfaces will accomodate them if we ever do decide to build them.
- We might introduce (seemingly) unnatural degrees of abstraction in our services to ensure that they leave us enough slack for future changes.

We must consider which of our interfaces will need to be how stable, and design our system with those stabilities in mind.